# Contributing to VFS for Git

Thank you for taking the time to contribute!

## Guidelines

\* [Code of Conduct](#code-of-conduct)

\* [Design Reviews](#design-reviews)

\* [Platform Specific Code](#platform-specific-code)

\* [Tracing and Logging](#tracing-and-logging)

\* [Error Handling](#error-handling)

\* [Background Threads](#background-threads)

\* [Coding Conventions](#coding-conventions)

\* [Testing](#testing)

## Code of Conduct

This project has adopted the [Microsoft Open Source Code of

Conduct](https://opensource.microsoft.com/codeofconduct/).

For more information see the [Code of Conduct

FAQ](https://opensource.microsoft.com/codeofconduct/faq/) or

contact [opencode@microsoft.com](mailto:opencode@microsoft.com)

with any additional questions or comments.

## Design Reviews

Architectural changes and new features should start with a design review. It's easier and wastes less time to incorporate feedback at this stage.

The design review process is as follows:

1. Create a pull request that contains a design document in Markdown (.md) format for the proposed change. Assign the `design-doc` label to the pull request.

2. Use the pull request for design feedback and for iterating on the design.

3. Once the design is approved, create a new issue with a description that includes the final design document. Include a link to the pull request that was used for discussion.

4. Close (without merging!) the pull request used for the design discussion.

## Platform Specific Code

- \*Prefer cross-platform code to platform-specific code\*

Cross-platform code is more easily reused. Reusing code reduces the amount of code that must be written, tested, and maintained.

- \*Platform specific code, and only platform specific code, should go in `GVFSPlatform`\*

When platform specific code is required, it should be placed in `GVFSPlatform` or one of the platforms it contains (e.g. `IKernelDriver`)

## Tracing and Logging

- \*The "Error" logging level is reserved for non-retryable errors that result in I/O failures or the VFS4G process shutting down\*

The expectation from our customers is that when VFS4G logs an "Error" level message in its log file either:

\* VFS4G had to shut down unexpectedly

\* VFS4G encountered an issue severe enough that user-initiated I/O would fail.

- \*Log full exception stacks\*

Full exception stacks (i.e. `Exception.ToString`) provide more details than the exception message alone (`Exception.Message`). Full exception stacks make root-causing issues easier.

- \*Do not display full exception stacks to users\*

Exception call stacks are not usually actionable for the user. Users frequently (sometimes incorrectly) assume that VFS4G has crashed when shown a full stack. The full stack \*should\* be included in VFS4G logs, but \*should not\* be displayed as part of the error message provided to the user.

- \*Include relevant details when logging exceptions\*

Sometimes an exception call stack alone is not enough to root cause failures in VFS4G. When catching (or throwing) exceptions, log relevant details that will help diagnose the issue. As a general rule, the closer an exception is caught to where it's thrown, the more relevant details there will be to log.

Example:

```

catch (Exception e)

{

EventMetadata metadata = new EventMetadata();

metadata.Add("Area", "Mount");

metadata.Add(nameof(enlistmentHookPath), enlistmentHookPath);

metadata.Add(nameof(installedHookPath), installedHookPath);

metadata.Add("Exception", e.ToString());

context.Tracer.RelatedError(metadata, $"Failed to compare {hook.Name} version");

}

```

## Error Handling

- \*Fail fast: An error or exception that risks data loss or corruption should shut down VFS4G immediately\*

Preventing data loss and repository corruption is critical. If an error or exception occurs that could lead to data loss, it's better to shut down VFS4G than keep the repository mounted and risk corruption.

- \*Do not catch exceptions that are indicative of a programmer error (e.g. `ArgumentNullException`)\*

Any exceptions that result from programmer error (e.g. `ArgumentNullException`) should be discovered as early in the development process as possible. Avoid `catch` statements that would hide these errors (e.g. `catch(Exception)`).

The only exception to this rule is for [unhandled exceptions in background threads](#bgexceptions)

- \*Do not use exceptions for normal control flow\*

Prefer writing code that does not throw exceptions. The `TryXXX` pattern, for example, avoids the performance costs that come with using exceptions. Additionally, VFS4G typically needs to know exactly where errors occur and handle the errors there. The `TryXXX` pattern helps ensure errors are handled in that fashion.

Example: Handle errors where they occur (good):

```

bool TryDoWorkOnDisk(string fileContents, out string error)

{

if (!TryCreateReadConfig())

{

error = "Failed to read config file";

return false;

}

if (!TryCreateTempFile(fileContents))

{

error = "Failed to create temp file";

return false;

}

if (!TryRenameTempFile())

{

error = "Failed to rename temp file";

if (!TryDeleteTempFile())

{

error += ", and failed to cleanup temp file";

}

return false;

}

error = null;

return true;

}

```

Example: Handle errors in `catch` without knowing where they came from (bad):

```

bool TryDoWorkOnDisk(string fileContents, out string error)

{

try

{

CreateReadConfig();

CreateTempFile(fileContents);

RenameTempFile();

}

catch (Exception ex) when (ex is IOException || ex is UnauthorizedAccessException)

{

error = "Something went wrong doing work on disk";

try

{

if (TempFileExists())

{

DeleteTempFile();

}

}

catch (Exception e) when (e is IOException || e is UnauthorizedAccessException)

{

error += ", and failed to cleanup temp file";

}

return false;

}

error = null;

return true;

}

```

- \*Provide the user with user-actionable messages whenever possible\*

Don't tell a user what went wrong. Help the user fix the problem.

Example:

> `"You can only specify --hydrate if the repository is mounted. Run 'gvfs mount' and try again."`

## Background Threads

- \*Avoid using the thread pool (and avoid using async)\*

`HttpRequestor.SendRequest` makes a [blocking call](https://github.com/Microsoft/VFSForGit/blob/4baa37df6bde2c9a9e1917fc7ce5debd653777c0/GVFS/GVFS.Common/Http/HttpRequestor.cs#L135) to `HttpClient.SendAsync`. That blocking call consumes a thread from the managed thread pool. Until that design changes, the rest of VFS4G must avoid using the thread pool unless absolutely necessary. If the thread pool is required, any long running tasks should be moved to a separate thread managed by VFS4G itself (see [GitMaintenanceQueue](https://github.com/Microsoft/VFSForGit/blob/4baa37df6bde2c9a9e1917fc7ce5debd653777c0/GVFS/GVFS.Common/Maintenance/GitMaintenanceQueue.cs#L19) for an example).

Long-running or blocking work scheduled on the managed thread pool can prevent the normal operation of VFS4G. For example, it could prevent downloading file sizes, loose objects, or file contents in a timely fashion.

- <a id="bgexceptions"></a>\*Catch all exceptions on long-running tasks and background threads\*

Wrap all code that runs in the background thread in a top-level `try/catch(Exception)`. Any exceptions caught by this handler should be logged, and then VFS4G should be forced to terminate with `Environment.Exit`. It's not safe to allow VFS4G to continue to run after an unhandled exception stops a background thread or long-running task. Testing has shown that `Environment.Exit` consistently terminates the VFS4G mount process regardless of how background threads are started (e.g. native thread, `new Thread()`, `Task.Factory.StartNew()`).

An example of this pattern can be seen in [`BackgroundFileSystemTaskRunner.ProcessBackgroundTasks`](https://github.com/Microsoft/VFSForGit/blob/4baa37df6bde2c9a9e1917fc7ce5debd653777c0/GVFS/GVFS.Virtualization/Background/BackgroundFileSystemTaskRunner.cs#L233).

## Coding Conventions

- \*Most C# coding style guidelines are covered by StyleCop\*

Fix any StyleCop issues reported in changed code. When adding new projects to VFS4G, be sure that StyleCop is analyzing them as part of the build.

- \*Prefer explicit types to interfaces and implicitly typed variables\*

Avoid the use of `var` (C#), `dynamic` (C#), and `auto` (C++). Prefer concrete/explicit types to interfaces (e.g. prefer `List` to `IList`).

The VFS4G codebase uses this approach because:

\* Interfaces can hide the performance characteristics of their underlying type. For example, an `IDictionary` could be a `SortedList` or a `Dictionary` (or several other data types).

\* Interfaces can hide the thread safety (or lack thereof) of their underlying type. For example, an `IDictionary` could be a `Dictionary` or a `ConcurrentDictionary`.

\* Explicit types make these performance and thread safety characteristics explicit when reviewing code.

\* VFS4G is not a public API and its components are always shipped together. Develoepers are free to make API changes to VFS4G's public methods.

- \*Method names start with a verb (e.g. "GetProjectedFolderEntryData" rather than "ProjectedFolderEntryData")\*

Starting with a verb in the name improves readability and helps ensure consistency with the rest of the VFS4G codebase.

- \*Aim to write self-commenting code. When necessary, comments should give background needed to understand the code.\*

Helpful (good) comment:

```

// Order the folders in descending order so that we walk the tree from bottom up.

// Traversing the folders in this order:

// 1. Ensures child folders are deleted before their parents

// 2. Ensures that folders that have been deleted by git

// (but are still in the projection) are found before their

// parent folder is re-expanded (only applies on platforms

// where EnumerationExpandsDirectories is true)

foreach (PlaceholderListDatabase.PlaceholderData folderPlaceholder in placeholderFoldersListCopy.OrderByDescending(x => x.Path))

```

Obvious (bad) comment:

```

// Check if enumeration expands directories on the current platform

if (GVFSPlatform.Instance.KernelDriver.EnumerationExpandsDirectories)

```

- \*Add new interfaces when it makes sense for the product, not simply for unit testing\*

When a class needs to be mocked (or have a subset of its behavior mocked), prefer using virtual methods instead of adding a new interface. VFS4G uses interfaces when multiple implementations of the interface exist in the product code.

- \*Check for `null` using the equality (`==`) and inequality (`!=`) operators rather than `is`\*

A corollary to this guideline is that equality/inequality operators that break `null` checks should not be added (see [this post](https://stackoverflow.com/questions/40676426/what-is-the-difference-between-x-is-null-and-x-null) for an example).

- \*Use `nameof(...)` rather than hardcoded strings\*

Using `nameof` ensures that when methods/variables are renamed the logging of those method/variable names will also be updated. However, hard coded strings are still appropriate when they are used for generating reports and changing the strings would break the reports.

### C/C++

- \*Do not use C-style casts. Use C++-style casts.\*

C++ style casts (e.g. `static\_cast<T>`) more clearly express the intent of the programmer, allow for better validation by the compiler, and are easier to search for in the codebase.

- \*Declare static functions at the top of source files\*

This ensures that the functions can be called from anywhere inside the file.

- \*Do not use namespace `using` statements in header files\*

`using` statements inside header files are picked up by all source files that include the headers and can cause unexpected errors if there are name collisions.

- \*Prefer `using` to full namespaces in source files\*

Example:

```

// Inside MyFavSourceFile.cpp

using std::string;

// Do not use `std::` namespace with `string`

static string s\_myString;

```

- \*Use a meaningful prefix for "public" free functions, and use the same prefix for all functions in a given header file\*

Example:

```

// Functions declared in VirtualizationRoots.h have "VirtualizationRoot\_" prefix

bool VirtualizationRoot\_IsOnline(VirtualizationRootHandle rootHandle);

// Static helper function in VirtualizationRoots.cpp has no prefix

static VirtualizationRootHandle FindOrDetectRootAtVnode(vnode\_t vnode, const FsidInode& vnodeFsidInode);

```

## Testing

- \*Add new unit and functional tests when making changes\*

Comprehensive tests are essential for maintaining the health and quality of the product. For more details on writing tests see [Authoring Tests](https://github.com/Microsoft/VFSForGit/blob/master/AuthoringTests.md).

- \*Functional tests are black-box tests and should not build against any VFS4G product code\*

Keeping the code separate helps ensure that bugs in the product code do not compromise the integrity of the functional tests.

### C# Unit Tests

- \*Add `ExceptionExpected` to unit tests that are expected to have exceptions\*

Example:

```

[TestCase]

[Category(CategoryConstants.ExceptionExpected)]

public void ParseFromLsTreeLine\_NullRepoRoot()

```

Unit tests should be tagged with `ExceptionExpected` when either the test code or the product code will throw an exception. `ExceptionExpected` tests are not executed when the debugger is attached, and this prevents developers from having to keep continuing the tests each time exceptions are caught by the debugger.

- \*Use a `mock` prefix for absolute file system paths and URLs\*

The unit tests should not touch the real file system nor should they reach out to any real URLs. Using `mock:\\` and `mock://` ensures that any product code that was not properly mocked will not interact with the real file system or attempt to contact a real URL.